2020 자율주행 교육

WeGo 위고 주식회사

목차

WeGo

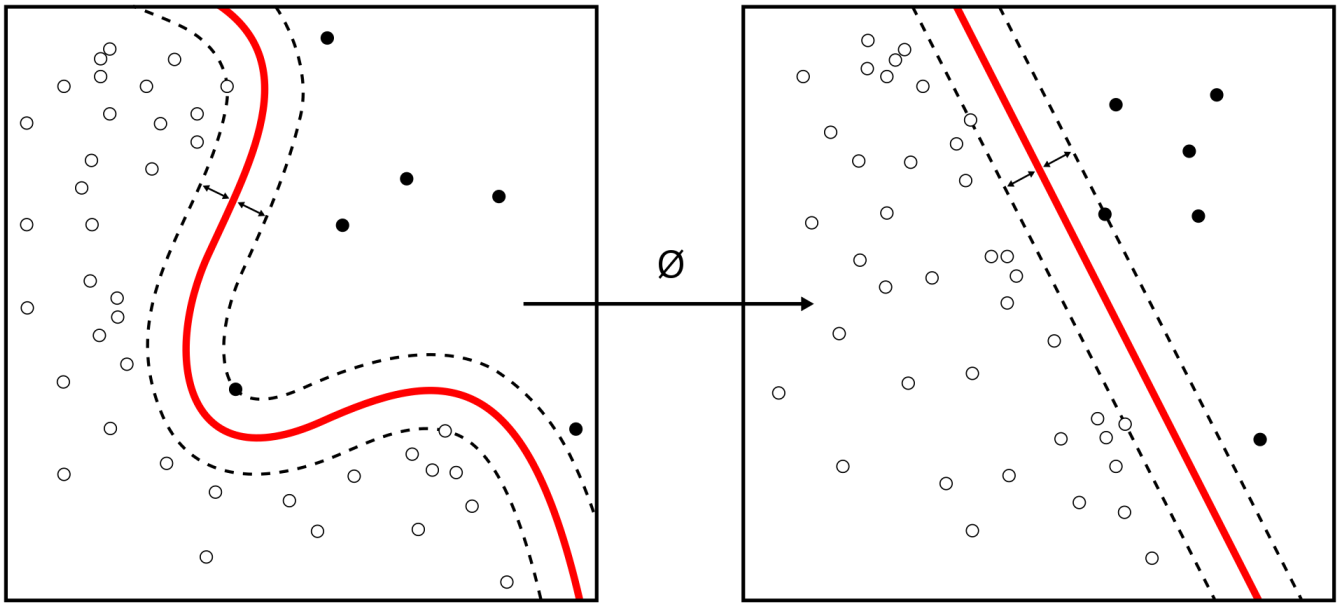
01

Machine Learning

WeGo

01. Machine Learning

- 인공지능의 한 분야
- 경험을 통해 성능을 개선하는 알고리즘
- SVM, K-NN, Deep Learning이 대표적
- Training, Evaluation, Test Data로 나누어서 진행
- 지도 학습, 비지도 학습, 자기 지도 학습, 강화 학습으로 나뉨



Support Vector Machine

WeGo

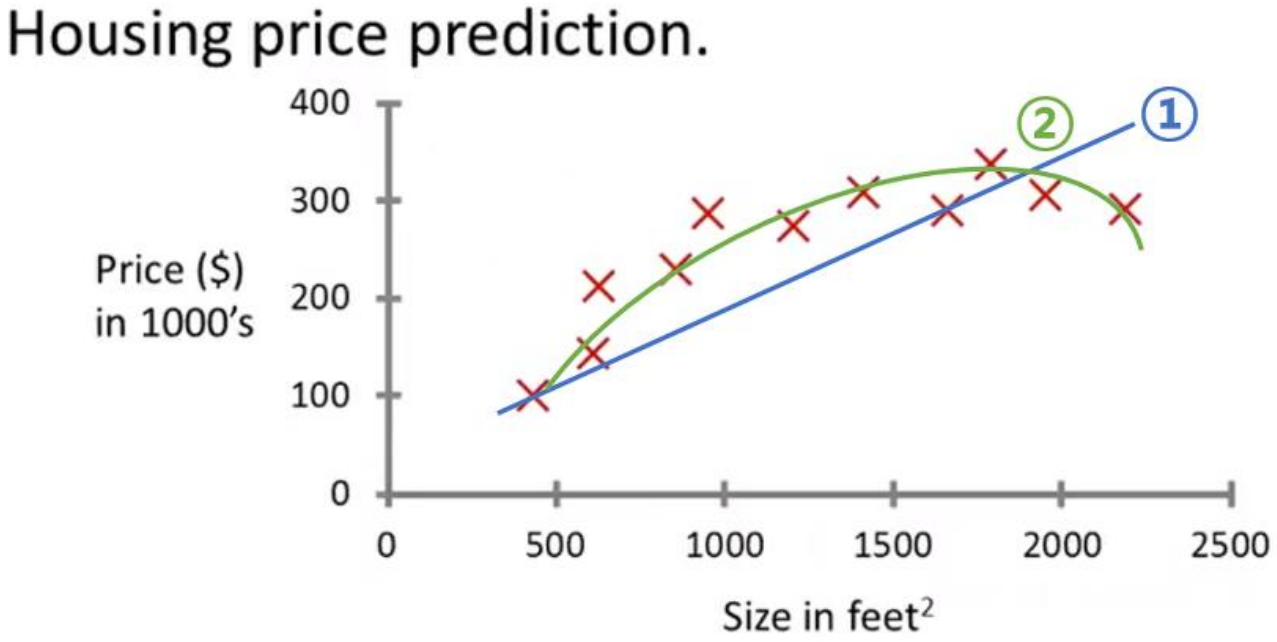
01. Machine Learning

- 지도 학습(Supervised Learning)

→ 특정 입력에 대해 정답이 있는 데이터가 주어질 경우에 사용하는 학습법

→ 정해진 Model에 따라 입력과 정답 사이의 관계를 기계가 직접 학습

→ 회귀분석, 분류에 사용이 가능

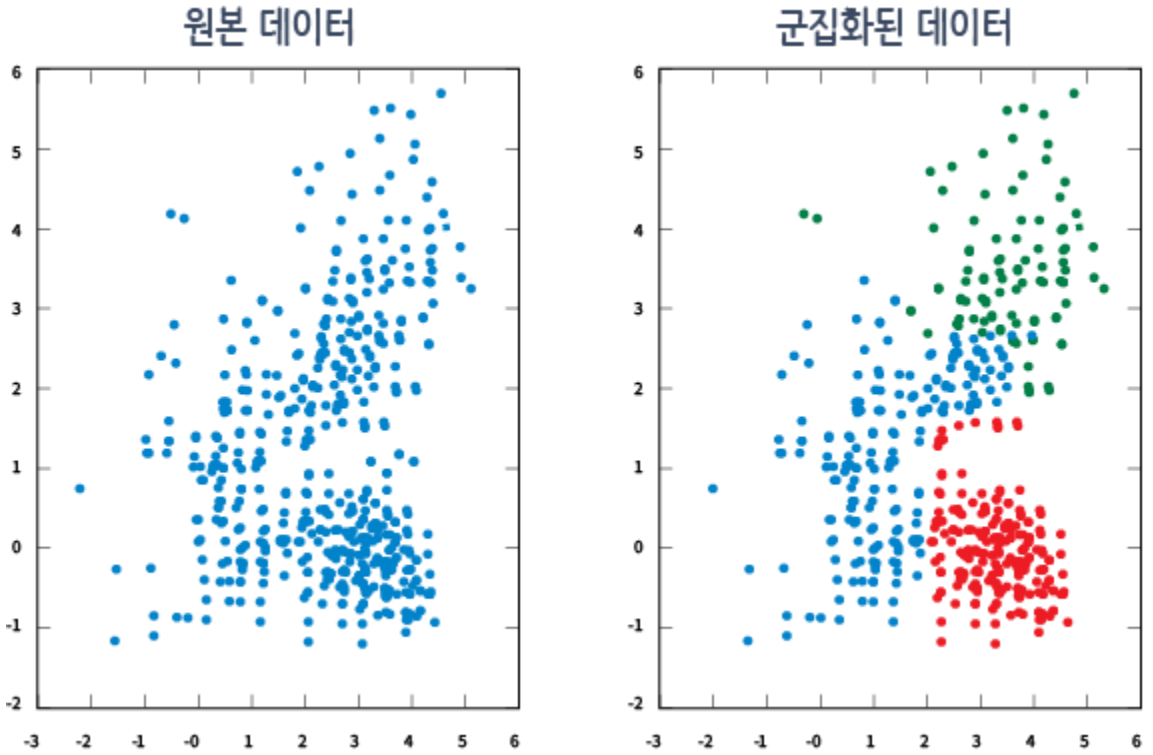


WeGo

- 비지도 학습(Unsupervised Learning)

→ 지도 학습과 반대로 특정 입력에 대해 정답이 없는 경우에 사용하는 학습법

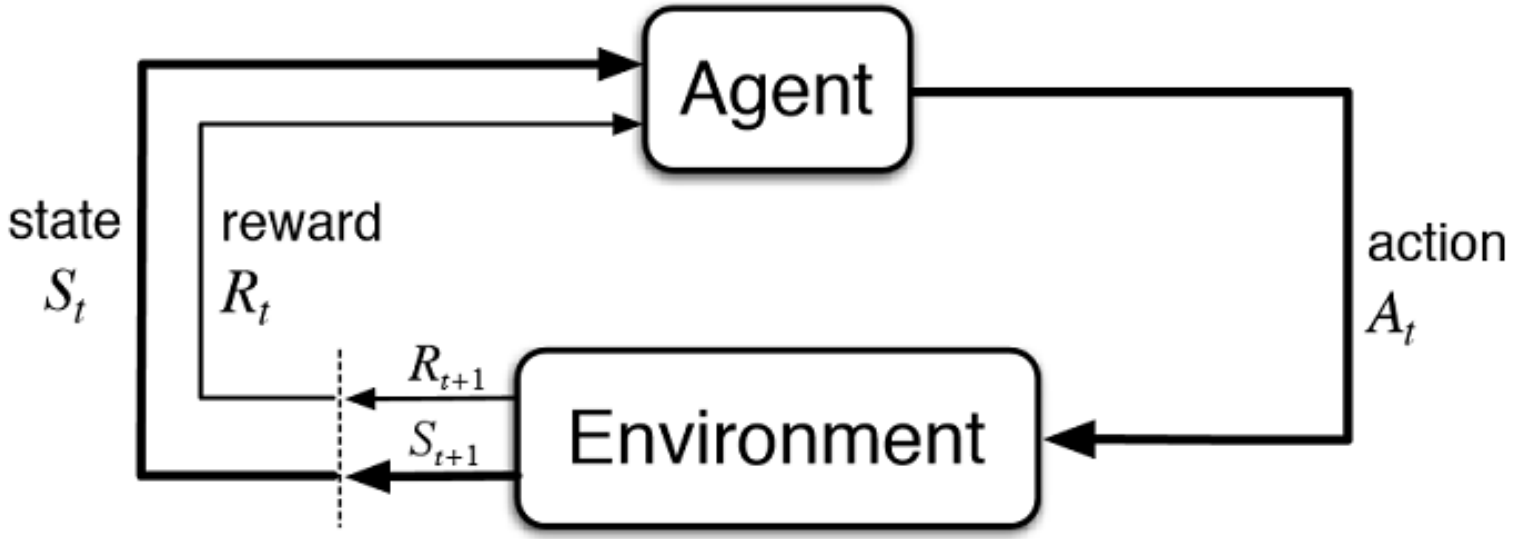
→ Clustering(군집화) - 유사한 성질끼리 묶어주는 것



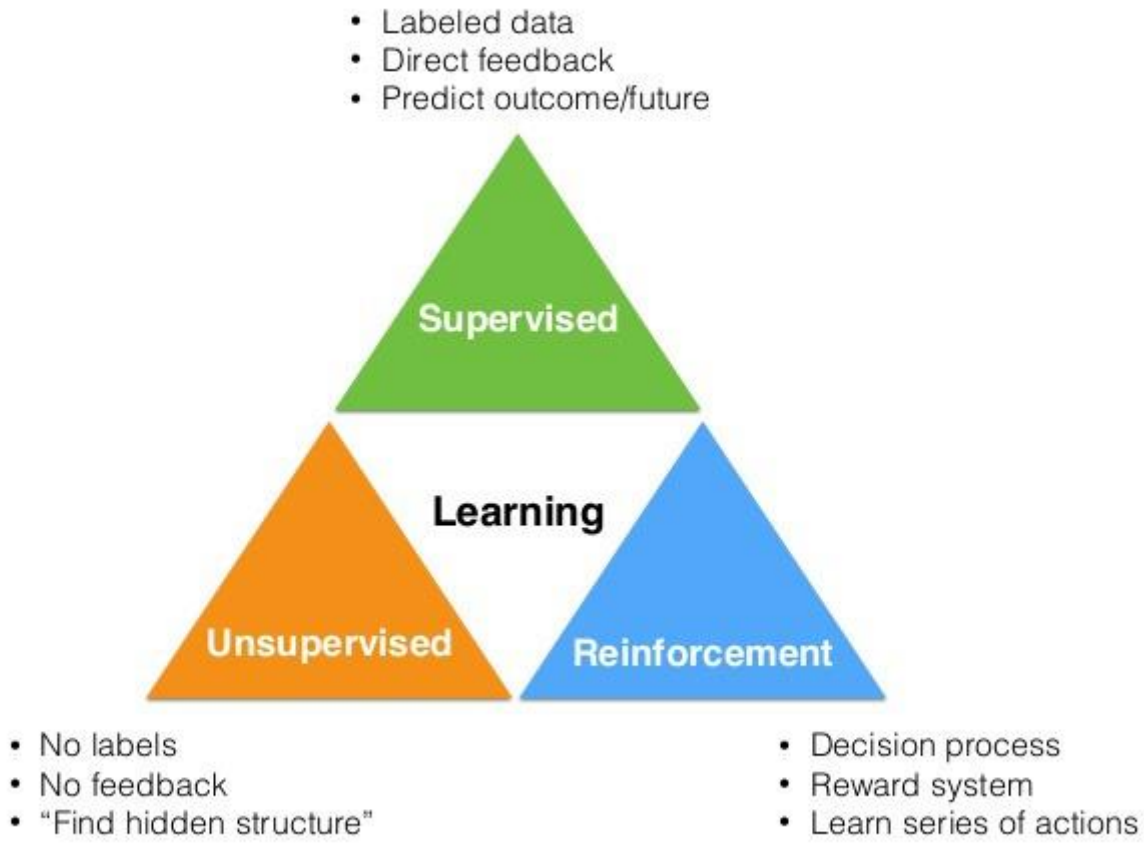
- 강화 학습(Reinforcement Learning)

→ Reward를 기준으로, 상을 최대화하고 벌을 최소화하는 학습 방식

→ Decision Process (특정 상태에 대해 선택하는 과정) 에 주로 사용



- Machine Learning의 종류 및 특징 정리

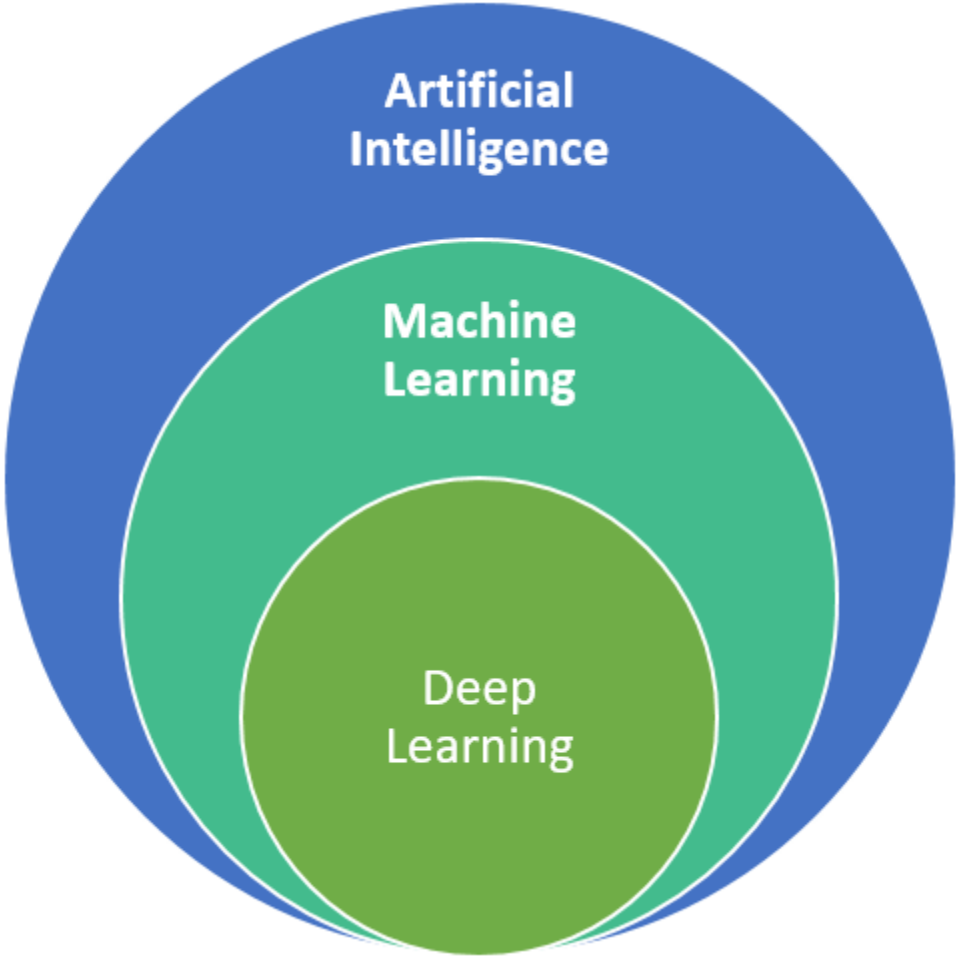


02

Deep Learning - Perceptron

WeGo

- AI, Machine Learning, Deep Learning의 상관 관계



02. Perceptron

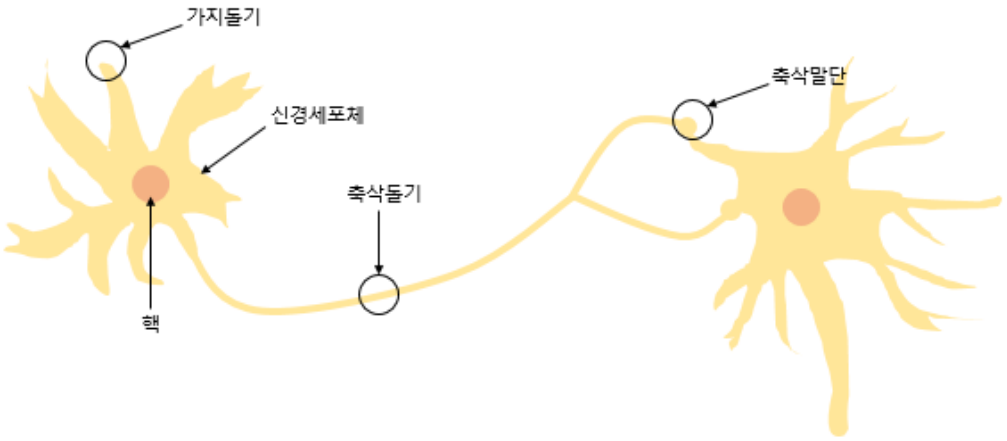
- Perceptron

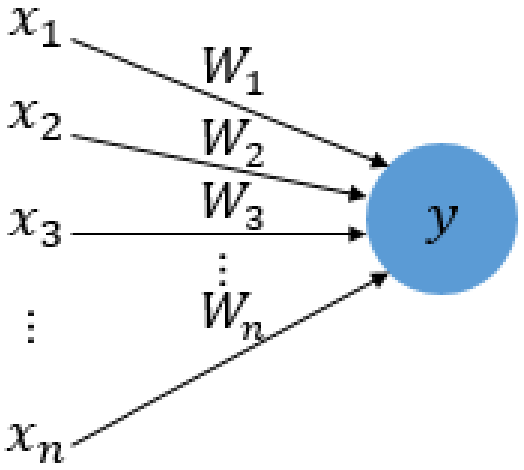
→ 1957년에 프랑크 로젠블라트가 제안한 초기 형태의 인공 신경망

→ 다수의 입력에서 하나의 결과를 내보내는 알고리즘

→ 실제 뇌를 구성하는 뉴런의 동작과 유사

→ 입력(가지돌기, x_n)이 일정 이상의 크기를 가지면 출력(축삭말단, y)



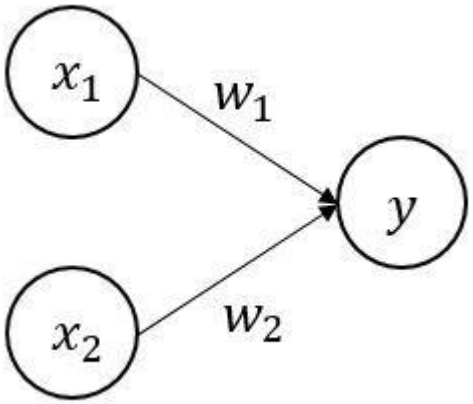


- Perceptron

→ 입력(x_1, x_2)과 가중치(w_1, w_2), 임계값(θ), 그리고 출력(y)

→ 각각의 원을 뉴런 또는 노드로 지칭

→ 가중치의 경우, 신호를 흐르거나 흐르지 않게 하므로, 저항과 유사한 역할



$$y = \begin{cases} 0, & (w_1x_1 + w_2x_2 \leq \theta \\ 1, & (w_1x_1 + w_2x_2 > \theta \end{cases}$$

- AND Gate

→ 퍼셉트론을 통해서 구현

→ 가중치와 임계값을 조절하여 AND Gate 구현

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

- AND Gate

→ 퍼셉트론을 통해서 구현

→ 가중치와 임계값을 조절하여 AND Gate 구현

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

```
def AND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.7
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    else:
        return 1
```

- 편향 (bias)

→ 임계값에 해당하는 θ를 -b로 치환하여, 퍼셉트론을 다음과 같이 변환 가능

$$y = \begin{cases} 0, & (w_1x_1 + w_2x_2 \leq \theta \\ 1, & (w_1x_1 + w_2x_2 > \theta \end{cases} \qquad y = \begin{cases} 0, & (b + w_1x_1 + w_2x_2 \leq 0 \\ 1, & (b + w_1x_1 + w_2x_2 > 0 \end{cases}$$

- AND Gate (bias 추가)

→ 퍼셉트론을 통해서 구현

→ 가중치와 임계값을 조절하여 AND Gate 구현

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

- AND Gate (bias 추가)

→ 퍼셉트론을 통해서 구현

→ 가중치와 임계값을 조절하여 AND Gate 구현

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

```
import numpy
def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    tmp = np.sum(w*x)+b
    if tmp <= 0:
        return 0
    else:
        return 1
```

- NAND Gate

→ NAND는 Not AND Gate이고, AND와 반대 출력값을 가진다.

→ 따라서 가중치와 편향의 부호를 반대로 해주면 된다.

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

- NAND Gate

→ NAND는 Not AND Gate이고, AND와 반대 출력값을 가진다.

→ 따라서 가중치와 편향의 부호를 반대로 해주면 된다.

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

```
import numpy
def NAND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([-0.5, -0.5])
    b = 0.7
    tmp = np.sum(w*x)+b
    if tmp <= 0:
        return 0
    else:
        return 1
```

- OR Gate

→ OR Gate의 경우는 입력의 합의 형태로 나타난다.

→ 가중치와 임계값(b)를 조절하여, 구현할 수 있다.

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

- OR Gate

→ OR Gate의 경우는 입력의 합의 형태로 나타난다.

→ 임계값(b)을 조절하여, 구현할 수 있다.

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

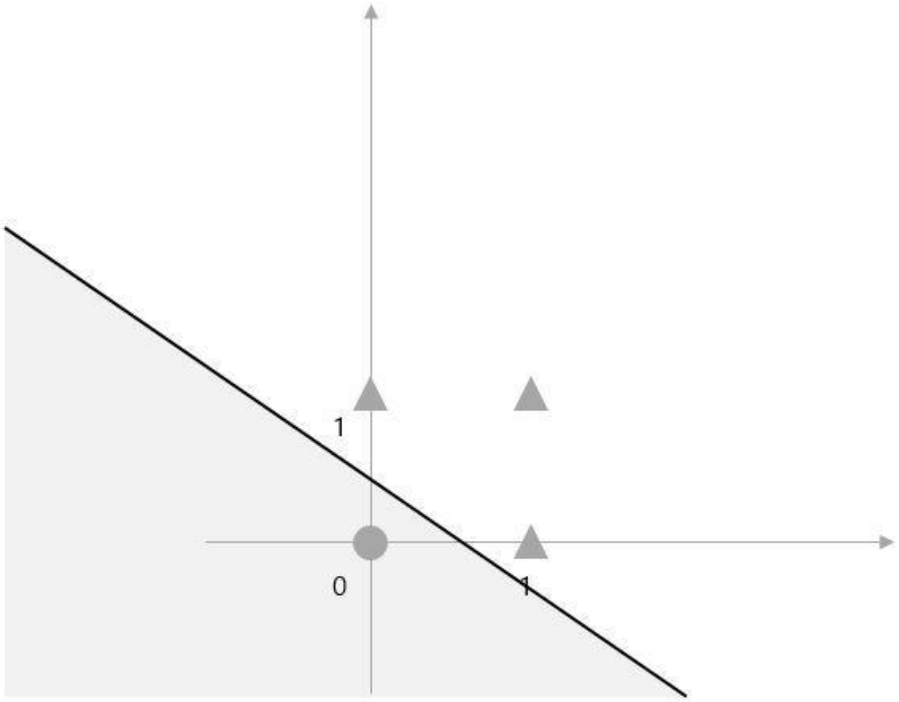
```
import numpy
def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.2
    tmp = np.sum(w*x)+b
    if tmp <= 0:
        return 0
    else:
        return 1
```

- 퍼셉트론의 한계

→ XOR Gate를 퍼셉트론을 이용하여 구현할 수 있을까?

→ 지금까지의 퍼셉트론을 이용해서는 XOR Gate를 구현할 수 없다.

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

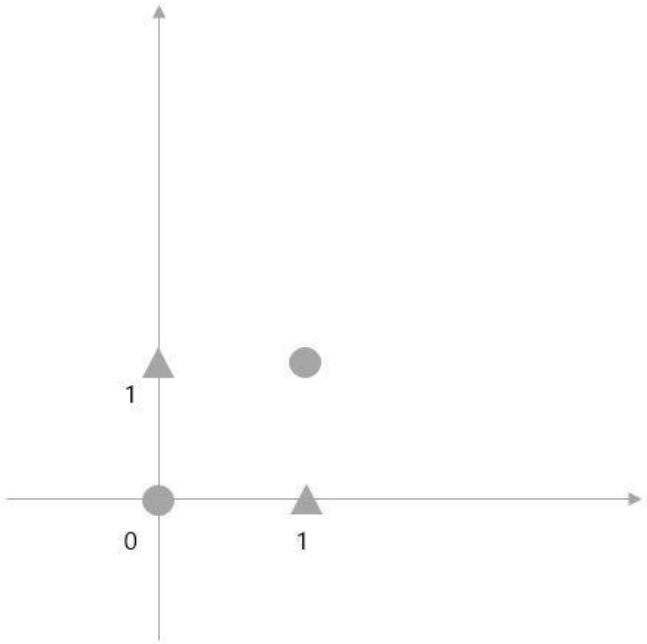


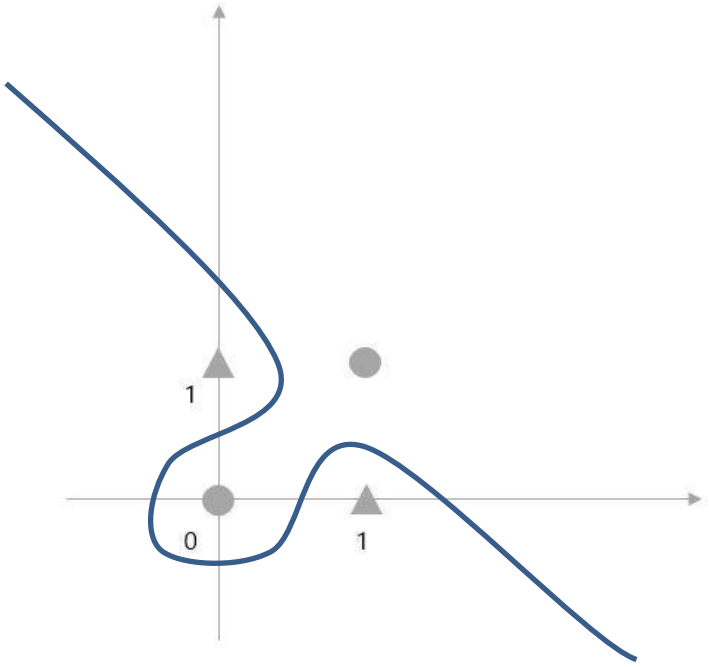
OR Gate

- 퍼셉트론의 한계

→ 하나의 직선을 이용해서는 XOR Gate를 구현하는 것이 불가능

→ 퍼셉트론은 직선 하나로 나눈 영역만 표현할 수 있다는 한계가 있음

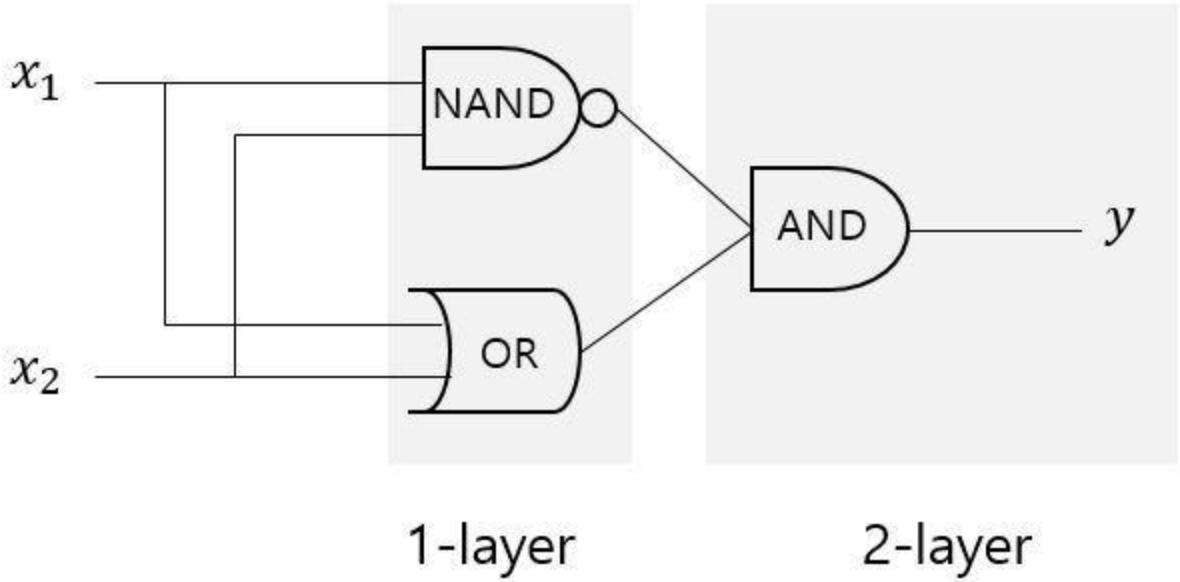




- 다층 퍼셉트론의 이용

→ XOR Gate를 구성하기 위해서는 이전의 AND, OR, NAND Gate의 결합을 이용

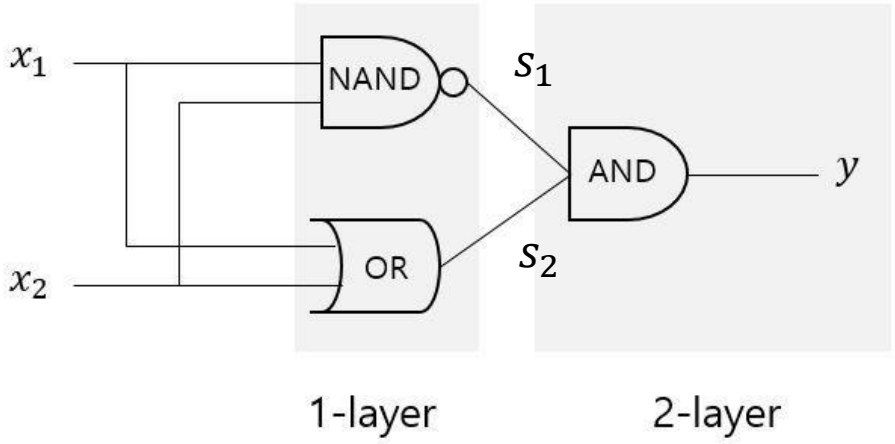
→ 세 개의 Gate를 결합하는 방식을 통해, XOR Gate를 구현 가능



- 다층 퍼셉트론의 이용

→ XOR Gate를 구성하기 위해서는 이전의 AND, OR, NAND Gate의 결합을 이용

→ 세 개의 Gate를 결합하는 방식을 통해, XOR Gate를 구현 가능

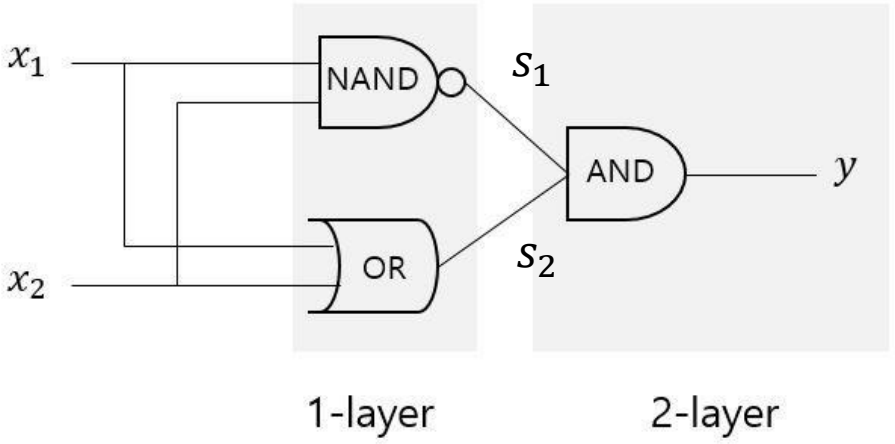


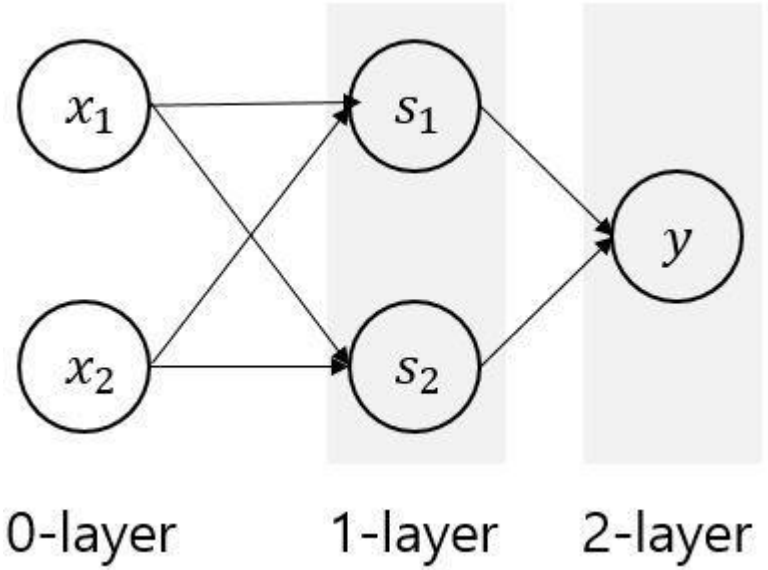

```
import numpy
def XOR(x1, x2):
    s1 = NAND(x1, x2)
    s2 = OR(x1, x2)
    y = AND(s1, s2)
    return y
```

- 다층 퍼셉트론의 이용

→ XOR Gate는 다층 구조 네트워크

→ 퍼셉트론을 여러 층 쌓는 방식을 통해 문제를 해결하는 방식이 Deep Learning





03

Deep Learning

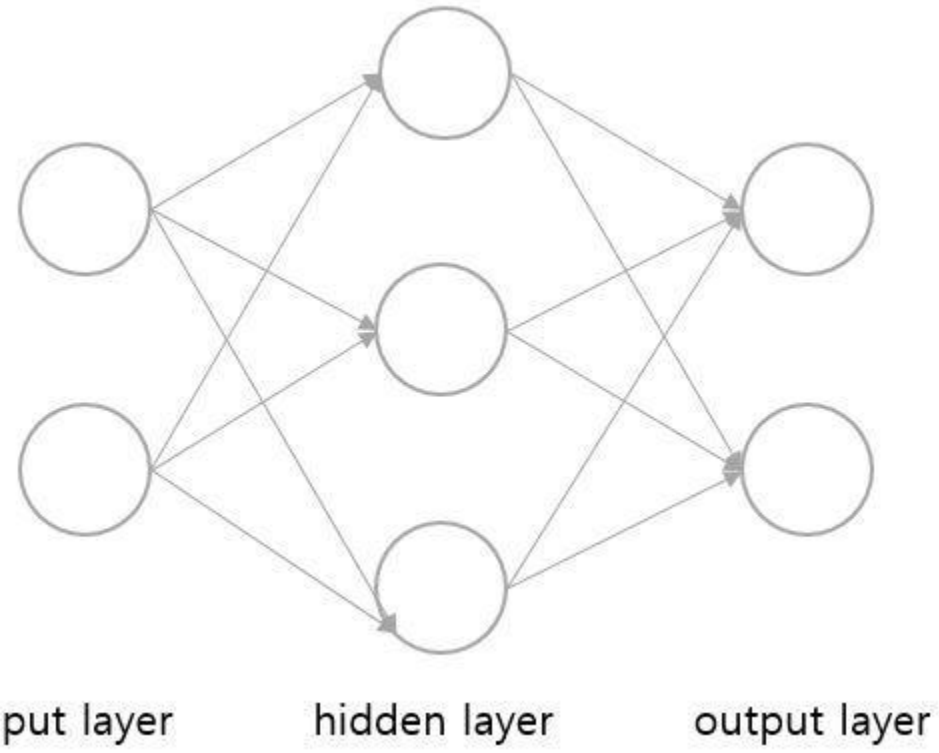
Neural Network

WeGo

- 신경망

→ 신경망은 입력층, 출력층, 은닉층으로 구성

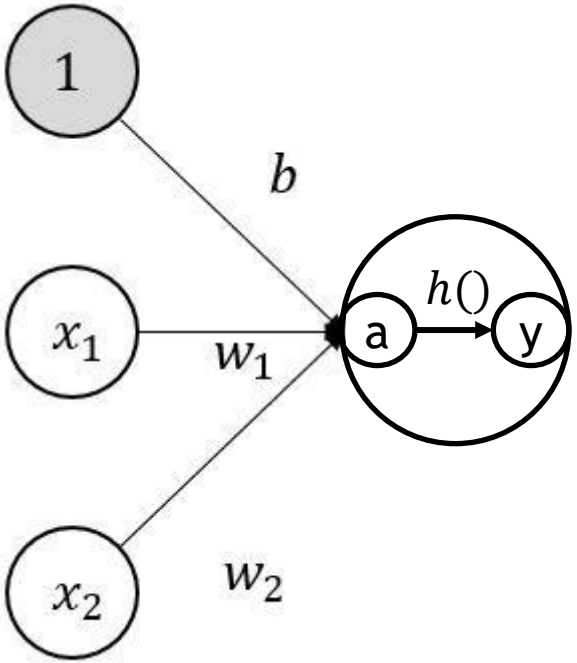
→ 사람에 따라 구성 층수를 기준으로 3층 또는 2층 신경망이라고 부를 수 있음

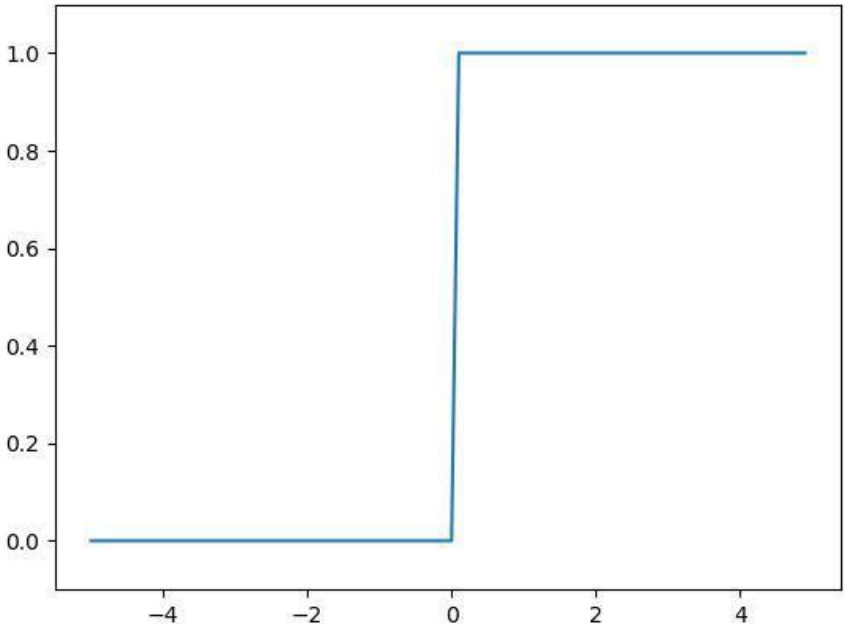


- 활성화 함수

→ 앞서 설명한 Perceptron의 경우 활성화 함수를 Step 함수로 사용

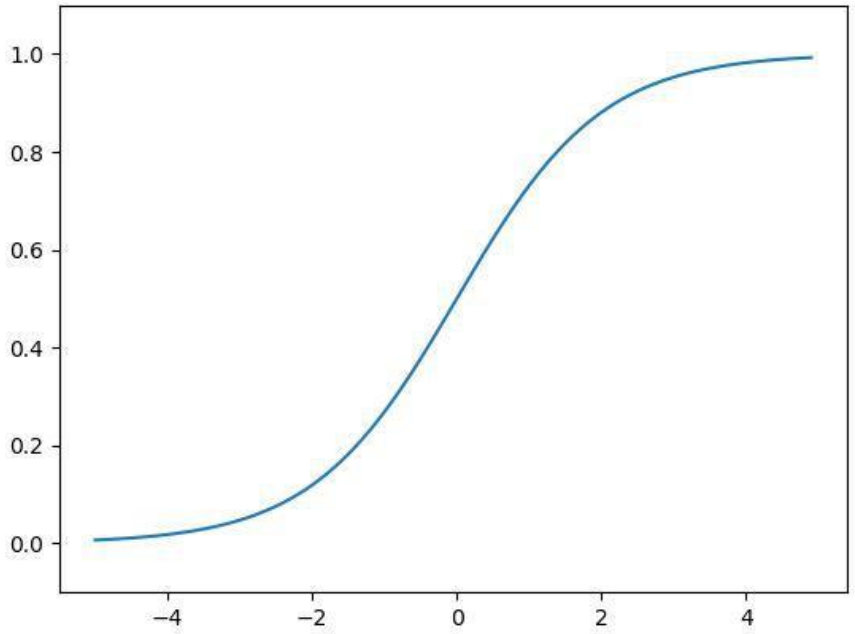
→ 실제로는 Step함수를 사용하지 않고, Sigmoid, Softmax, ReLU 함수 등을 사용





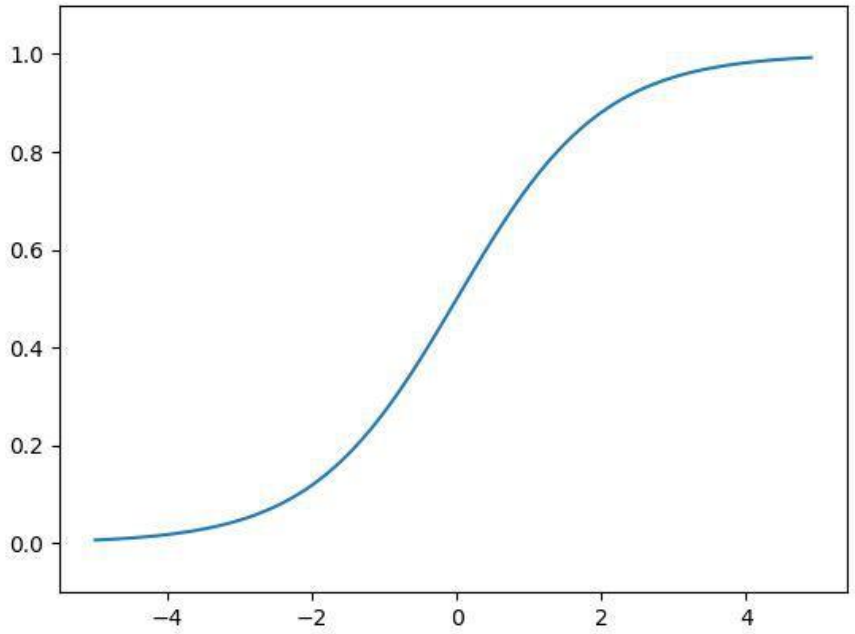
- Sigmoid function

$$h(x) = \frac{1}{1 + \exp(-x)}$$



- Sigmoid function

$$h(x) = \frac{1}{1 + \exp(-x)}$$




```
import numpy
import matplotlib.pylab as plt
def sigmoid(x):
    return 1/(1+np.exp(-x))

X = np.arrange(-5, 5, 0.1)
Y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1)
plt.show()
```

- 활성화 함수를 비선형을 사용하는 이유

→ 활성화 함수를 선형으로 사용할 경우 신경망을 깊게 하는 의미가 없어짐

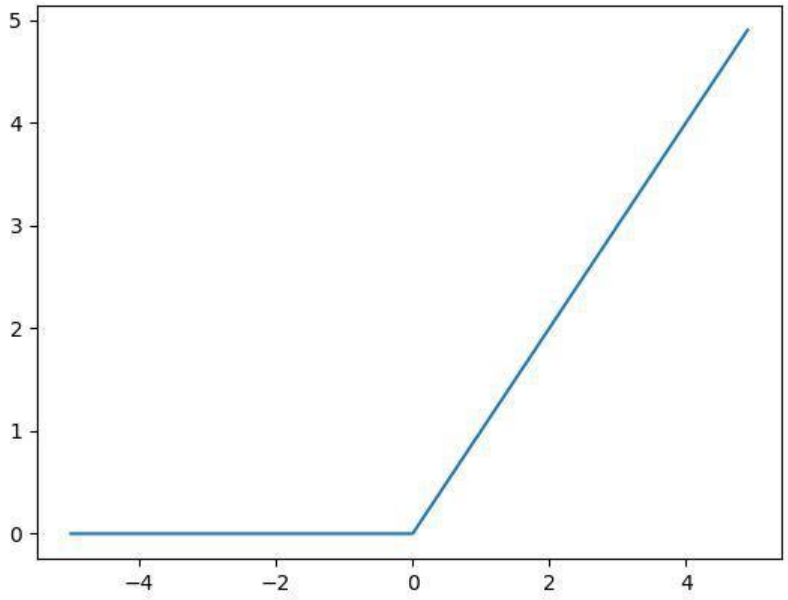
Ex) $h(x) = cx$를 활성화 함수로 사용할 때, 3층일 경우

$y = h\left(h\left(h(x)\right)\right) = c * c * c * x = c^3x$가 된다.

이는 $h(x) = c^3x$의 하나와 동일한 역할을 하게 되므로, 의미가 없어진다.

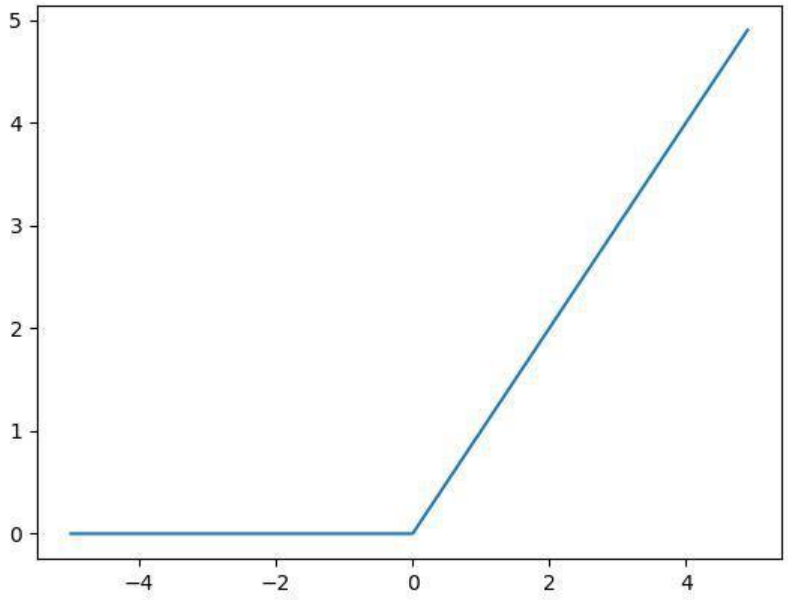
- ReLU function

$$h(x) = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases}$$



- ReLU function

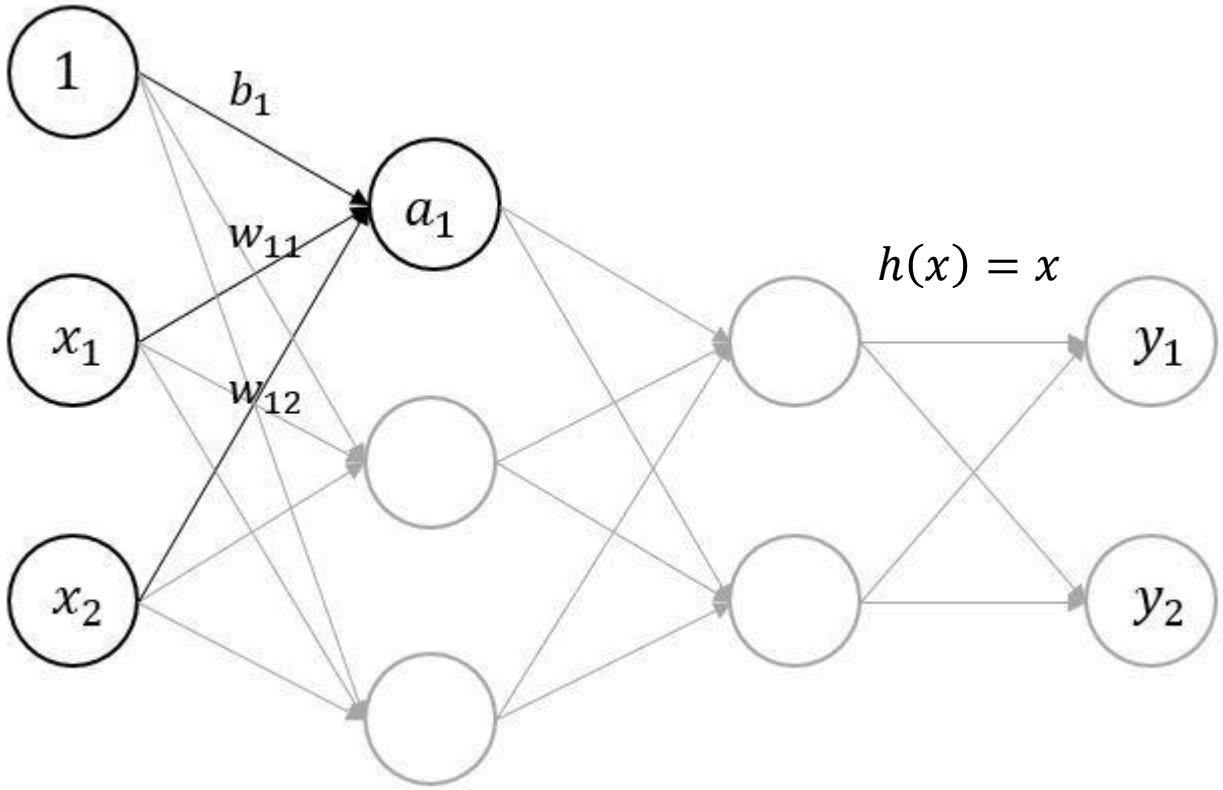
$$h(x) = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases}$$




```
import numpy
import matplotlib.pylab as plt
def relu(x):
    return np.maximum(0, x)

X = np.arrange(-5, 5, 0.1)
Y = relu(x)
plt.plot(x, y)
plt.ylim(-0.1, 5)
plt.show()
```

- 순전파(Feed Forward Propagation)



```
def identify_function(x):
    return x
def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
    network['b3'] = np.array([0.1, 0.2])
    return network
def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)
    return y
network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
```

WeGo